AFIT/GCS/ENG/92D-17

AD-A259 220

A SYNTHETIC ENVIRONMENT FLIGHT SIMULATOR
THE AFIT VIRTUAL COCKPIT

THESIS

John C. Switzer
Captain, USAF

AFIT/GCS/ENG/92D-17

DTIC
S ELECTE D
JAN 14 1993
E

Approved for public release; distribution unlimited

93-00092

93 1 1 042

AFIT/GCS/ENG/92D-17

A SYNTHETIC ENVIRONMENT FLIGHT SIMULATOR: THE AFIT VIRTUAL COCKPIT

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science Computer Engineering

DTIC QUALITY INSPECTED 5

John C. Switzer, B.S.E.E.
Captain, USAF

December, 1992

Accesion For
NTIS CRA&I ☒
DTIC TAB □
Unannounced □
Justification ....
By ....
Distribution /
Availability Codes

Dist	Avail and / or Special
A-1	

Approved for public release; distribution unlimited

Acknowledgments

I am deeply indebted to two people for their help on this project. First, Lt Col Amburn, my thesis advisor, for his absolutely tireless assistance with the *Virtual Cockpit* implementation. He is probably the busiest man I have ever known, yet he was never too busy to spend as much time with me as I needed to get over whatever hurdle the day had in store for me.

A huge amount of gratitude to my wife, Kellie. She helped me in ways she will probably never realize. Her most tangible support came in the way of proofing this paper for me. She has a beautiful grasp of the written language and spent hours pouring over every detail of this paper, looking for misspelled words, verb-tense disagreements, ambiguities, and "stuff that just didn't make sense." This paper wouldn't have been half as good without her red pen. She also had to put up with my irregular, and very full, schedule for a year and half. She never complained about it -- instead she made things as easy as possible for me.

And I can't leave out my daughter, Sarah, and son, Danny, for their very mature acceptance of "Daddy's busy." I didn't ever seem to have any time to play with them, and somehow they understood that.

Also, I'd like to express my appreciation to DARPA and Maj Dave Neyland for their commitment to AFIT and synthetic environment research.

John C. Switzer

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/92D-17

Abstract

This thesis describes the design and implementation of a part of the *Virtual Cockpit*: a synthetic environment, distributed network flight simulator. The goal of the project was to prove the concept that this type of flight simulator could fill the gap between high-end, very expensive flight simulators and low-end game quality flight simulators.

Discussed are: object-oriented design techniques, multi-processor utilization, the flight dynamics model, synthetic environment technology, the frame-rate vs. realism issue, and the interfaces to a realistic joystick and throttle.

A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit

I. Introduction

1.1 Background

The Air Force Institute of Technology (AFIT) Graphics Lab has pursued research in the synthetic environment arena since 1988. Research continues because this field holds much promise for the future of flight simulation and pilot task training. As machine speeds and graphic update rates continue to increase, so does the potential for dramatic realism.

This thesis extends the synthetic environment research to develop a realistic, distributed network, synthetic environment flight simulator which serves as both a proof of concept and as a platform for further research. We designated the final product the *AFIT Virtual Cockpit.*

AFIT had an operational synthetic environment flight simulator before now (Simpson, 1991), but while it was a good first step, it lacked the needed fidelity. The head-mounted display (HMD) resolution was poor, the workstation it ran on was slow by today's standards, the joystick and throttle were simple game joysticks, and the flight dynamics model was faulty.

By reusing many C++ classes from last year, along with significant Defense Advanced Research Projects Agency (DARPA) funding, we were able to improve on every aspect of the previous simulator. The *Virtual Cockpit* uses a much faster computer with multiple processors and texture mapping capabilities, a higher resolution HMD, and a realistic aircraft-type joystick and throttle.

1.2 Problem Statement

There exists a gap between the high-end, very realistic flight simulators used to train pilots how to fly a particular airplane, and the low-end personal computer or workstation flight simulators used primarily as games. The Aii Force needs a relatively cheap but realistic flight simulator that will fill the void between these two extremes and train pilots in certain tasks. Specifically, the simulator needs to be of high enough fidelity to provide a believable task training platform, yet cost less than $250,000 — more than two orders of magnitude reduction in price from today's high-end simulators. (Neyland, 1992)

Since a synthetic environment flight simulator does not have a motion system to simulate G-forces, believability must be achieved with the visual system alone. Since vision plays a large role in the equilibrium of the pilot, a realistic image of the world overcomes many of the body's perceptions of motion. (Rolfe, 1986: 17)

The specific tasks addressed by the *Virtual Cockpit* do not necessarily require an expensive replica of a certain airplane or a moveable platform to simulate motion and acceleration. Instead, a merely believable airplane is needed so that task training can be accomplished more efficiently. (Neyland, 1992)

Task training includes such areas as searching for mobile ground targets, RADAR techniques, weapon systems training, multi-aircraft maneuvers, emergency procedures, and many others. While conventional flight simulators can be used to train pilots in these tasks, they are much too expensive and unwieldy to do so economically. The expense and space requirements for high-end flight simulators preclude using them for training exercises that require more than one or two simulated airplanes.

1.3 Scope

The goal of this thesis effort was to create a realistic, distributed network, synthetic environment flight simulator. My efforts were focused on the overall design of the system, the design of the cockpit and cockpit instrumentation, the flight dynamics, and the hardware interfaces (joystick, throttle, rudder pedals, and head tracking). This thesis does not attempt to document the entire project. It will only deal with the third that I implemented. Two other students worked with me. Capt Dean McCarty (McCarty, 1993) concentrated on the database management and graphics rendering, and Capt Steven Sheasby (Sheasby, 1992) worked on the simulator's connection to a distributed network. Please see these other two theses for a more complete understanding of the entire project.

The *Virtual Cockpit* was included in six Department of Defense exercises (Zealous Pursuits 1-6) that took place in October through December of 1992. It joined numerous other simulated vehicles, via DARPA's distributed simulation network, in a simulated air and ground battle based on the closing weeks of Desert Storm.

1.4 Motivation for the Virtual Cockpit

1.4.1 Low Cost A conventional high-end flight simulator is very economical considering the role it plays. It replaces an aircraft and does so very realistically. It is safer to use than a real aircraft, it is cheaper to maintain, it uses no fuel, and can be used more frequently. (Rolfe, 1986: 234-235) However, to use it for task training, a function that doesn't require the highest fidelity, is cost prohibitive. A high-end flight simulator can cost more than $30,000,000. Our synthetic environment flight simulator was produced for under $250,000 — a price low enough to warrant using many of them for a single training exercise.

1.4.2 Minimal Hardware Required A conventional flight simulator is made up completely of proprietary hardware. It requires dedicated, specific computers, a cockpit that is fully equipped with real avionics gear, a moveable platform with which to simulate motion and acceleration, and some kind of visual system that surrounds the entire cockpit — usually a dome. (Rolfe, 1986: 8-10)

A synthetic environment flight simulator can take up to one hundredth of the space of a conventional high-end flight simulator (which frequently needs its own building) and it requires very little hardware.

1.4.3 Easily Reconfigurable Probably the biggest advantage a synthetic environment flight simulator has over the conventional flight simulator is the ease with which changes can be made to it. The entire simulator can be transformed into a new airplane quickly and inexpensively. It can model many different aircraft (including those that have not yet been designed.) It can serve as a test platform for new avionics and weapon systems. It can be modified cheaply to keep pace with technological advances by merely upgrading the computer and/or the visual system. Most, if not all, of these modifications are not possible with conventional simulators — there is just too much dedicated hardware involved.

1.4.4 New Systems Development New aircraft subsystems can be easily prototyped and tested in the *Virtual Cockpit*. Instead of going to the expense of prototyping the actual hardware and integrating it into the aircraft for testing, a virtual system can be readily added to the synthetic environment simulator using the same operational code the final product will use. Changes can be made very quickly to both the virtual hardware and the operational characteristics, saving millions of dollars and years in development time.

1.5 Hardware/Software

Our equipment was all off-the-shelf and included:

- Silicon Graphics Iris 440 VGXT Workstation
- Polhemus "Looking Glass" Fiber-Optic Head-Mounted Display (Prototype)
- Thrust Master WCS Stick, Throttle, and Rudder Pedals
- Polhemus Fast Tracker Position Reporting System

We also used the following software for the project:

- C++, (AT&T): An object-oriented programming language lends itself nicely to a program that models a real-world system.(Booch, 1991)
- Multigen (Software Systems, Santa Clara, CA): Multigen is a modeling tool that allowed us the capability to design a detailed F-15E, texture map the instruments, and place culture (trees, buildings, etc.) into our virtual world.

1.6 Approach and Methodology

Keeping sound software engineering principles in mind, I used an object-oriented approach to both the design and implementation. I studied Capt Simpson's design (Simpson, 1991), which was also object-oriented, and took many of the C++ classes from his effort and incorporated them into mine with no modifications. Most of the reused classes were low level device drivers — something that C++ models well.

1.6.1 Object-Oriented Design Object-oriented design and implementation were used in the development to facilitate reusability and ease of change. This was especially important for this project since it will be used for research and will be modified every year. Future researchers will want to take out whole systems (like the RADAR) and replace them with improved versions. Object-oriented design makes that an almost trivial task as long as the interfaces to the classes were designed well.

In designing the simulator I adhered to the following principles:

Modifiability If the design is to serve as a foundation for research, then it must be easy to change.

Extensibility The design of the program or system should be easily altered. The design must be constructed in such a manner that future additions to the design can be made with a minimum of effort. *Adding* to the design should not mean *redesigning* it from scratch.

Reusability The design should provide a format for reusing existing code and a framework that enables the inclusion of new code.

(Simpson, 1991)

1.6.2 Incremental-Build An incremental-build approach seemed to be the most practical software development methodology for this system. Since the simulator will probably never be complete, it was decided to produce the most functionality in the least amount of time. We had a first-cut system up and running in a matter of weeks, making extensive use of existing classes. From there we added functionality by adding new systems to the airplane and improving those already in place.

1.7 Thesis Overview

The following chapter presents an overview of synthetic environments, and object-oriented design methodology. Chapter 3 discusses the design of the virtual cockpit, and the last chapter presents research results and recommendations for further study.

II. Background

2.1 Introduction

This chapter will provide some background to put this thesis effort in perspective. Covered will be: a brief history of flight simulators, the recent advances in synthetic environment technology, an introduction to object-oriented design, and some of the previous research that has resulted in similar synthetic environment systems.

2.2 Flight Simulators

"The essential form of flight simulation is the creation of a dynamic representation of the behaviour of an aircraft in a manner which allows the human operator to interact with the simulation as a part of the simulation" (Rolfe, 1986: 3). Justice cannot be done to the long and complex history of flight simulators in this document, but a brief overview will serve to place recent technological advances in perspective.

2.2.1 Background Flight simulators date back to 1910. They were originally designed and are still used today for teaching pilots to fly airplanes. They lend themselves to the kinds of mistakes new pilots make and are much more forgiving than real airplanes. (Rolfe, 1986)

With the advent of computer technology in the 1950s, the flight simulator became controlled by special purpose (typically analog) computers. Even today, most simulators use many special purpose digital computers to control subsystems of the simulator like the flight dynamics, RADAR, and image generation. (Rolfe, 1986: 33)

2.2.2 The Visual System The conventional high-end simulator of today is mounted on a moveable platform to simulate motion and acceleration. These aspects of the simulation are important when a pilot is learning to control an airplane. Sight, however, remains the biggest contributor to the human sense of orientation. It is, therefore, more essential to have a good visual system than a motion system. (Rolfe, 1986: 17)

The visual component of the flight simulator has always lagged behind the other systems. It was not until the 1950s and the advent of computers that there was any real visual system at all.(Rolfe, 1986: 33) From then up to the 1980s, the visual systems were so crude that the only realistic outside scene was a nighttime view relying only on point light sources.(Clark, 1992) Today, with computers getting faster every year, it is possible to model terrain and cultural features with polygons rather than points. Faster image generation means smooth realistic motion, and the ability to render more polygons every second allows us to model the environment with more detail.(Clark, 1992)

> In the development of new techniques, particularly those associated with the representation of the visual world using computer generated graphics, ...representing texture and accurate patterns of light and shadow is clearly important. At the same time the advent of the micro computer and its ready adoption as an adult toy has resulted in the creation of flight simulator game packages that offer recreation and challenge but do not set out to teach the user to fly. (Rolfe, 1986: 3)

2.3 Synthetic Environments

It was Ivan Sutherland who first conceptualized the idea of the *Ultimate Display* in 1965. His ideas are still far from reality — virtual or otherwise. He envisioned computer generated chairs that could be sat upon, computer generated apple pies that would look, smell, and taste like Mom's, and bullets that would be fatal.(Sutherland, 1965: 506) He defined the current synthetic environment paradigm, and he was also one of the first researchers to use computer graphics in a flight simulator.(Clark, 1992: 153)

2.3.1 The Head-Mounted Display Many different HMDs have been designed since Ivan Sutherland first proposed the idea and then created his own. They typically use two small CRT or LCD displays positioned in front of the eyes and anchored on a shelf of some sort. The head position is monitored by a magnetic sensor (like the Polhemus Navigation Sciences magnetic tracker). The field of view ranges from 25º to 120º horizontally. AFIT has been involved with this technology and has developed an HMD that uses two 3" color CRTs.(Chung, 1989: 45)

2.4 Previous AFIT Research In Synthetic Environments

Since 1988, AFIT has been researching synthetic environment technology. There have many implementations of the research and several supporting efforts.

2.4.1 The AFIT HMD Captain Rebo (Rebo, 1988) opened AFIT's synthetic environment research efforts by developing a head-mounted display for the graphics lab. His design included two small LCD TVs attached to a bicycle helmet with lenses inset to focus the eye at just a few inches from the face. He used the Polhemus 3-Space tracker to track the head position. His design was in use until this year.

2.4.2 A Battle Visualization Tool AFIT's first synthetic environment application was developed by Capt Lorimor (Lorimor, 1988). Using the HMD designed by Captain Rebo, he was able to display 3-D images of many aircraft moving through the environment. The time-stamped aircraft data came from recorded Red Flag exercises and was "played back" in real-time. The user was able to watch the engagement from any vantage point in the environment, including any one of the aircrafts' cockpits.

2.4.3 Software Support For The Synthetic Environment Capt Filer (Filer, 1989) continued building the groundwork for many applications to come. He made improvements to the HMD and developed a suite of software routines to control the HMD, Polhemus 3-Space Tracker, VPL Data Glove, Dimension Six Force-Torque Ball, and a joystick.

2.4.4 An ATO Visualization Tool Building on the above work, Capt Wardin (Wardin, 1989) developed a prototype system to visualize an Air Tasking Order (ATO). With the system, an ATO could be played out with aircraft, ground vehicles, and troop formations moving about in a miniature battle field. A user could watch a predicted battle unfold as aircraft bombed targets and ground forces (friendly and enemy) moved about. Threat regions around missile sites were identified with transparent 3-D structures, giving the user a very detailed look at usually intangible factors in a battle.

2.4.5 The Data Glove Capt Gerken (Gerken, 1991: 12) developed a state-based model to interpret the inputs from a VPL Data Glove. Hand gestures had been interpreted in past AFIT applications, but only static ones. Capt Gerken added the capability to use the motion associated with gestures to tremendously expand the "vocabulary" of the Data Glove.

2.4.6 Flight Simulators In A Synthetic Environment In the thesis effort most closely linked to this one, AFIT students designed and implemented a synthetic environment flight simulator in 1991. That simulator was also derived from Silicon Graphics' Flight program and ran on an Iris workstation using the AFIT HMD.

Although the design was sound, the flight dynamics model was faulty. It also suffered from a poor resolution HMD, a relatively slow graphics workstation (less than 5 frames/second), and a makeshift throttle and stick. The design was such that the simulator could be "disassembled" to the object level and put back together in another way. Many of the low-level objects are included in the current design with no modification.

2.5 Object-Oriented Design

"Design reuse has more potential for increasing the productivity of software development and maintenance than do traditional approaches to software reuse that emphasize reuse of smaller components. Current software development methods do not promote design reuse."(Spicer, 1990: xiii) There isn't much documentation yet on how much object-oriented design (OOD) has helped the software community to reuse previous designs and increase productivity, but agreement is almost unanimous that it certainly helps when programming in the small. For example: data structures, device controllers, device simulators, etc., can be defined and grouped in a fairly natural way.(Booch, 1987: 45-112)

OOD is at its best when modeling physical devices. Objects correspond very well to things, like joysticks, where information about the particulars of the device is hidden and certain fundamental operations are defined to act on the object. The small systems can then be combined into larger systems until one system is completely composed of subsystems. This approach was taken to its logical conclusion by the Software Engineering Institute (SEI). Engineers there have designed a complete flight simulator using OOD.(Lee, 1989) Every major system of the flight simulator was modeled. The systems include the engine and related subsystems, the flight controls and their subsystems, etc. The detail to which their design goes is quite a bit more thorough than was needed for the *Virtual Cockpit.* I concentrated on modeling only the largest systems and making reasonable design tradeoffs to increase the speed of the simulation.

Object-oriented designs are meant to be reusable. That was put to the test with this thesis effort. Many of the hardware interface objects designed by Capt Simpson last year(Simpson, 1991) were used this year. With that foundation, a completely different design was possible without modifying these objects. Capt Simpson concentrated on making his design modifiable. "If the design is to serve as a foundation for research, then it ought to be easy to change."(Simpson, 1991: 1-2)

Booch states that there are five attributes of complex systems:

1. The system will possess a hierarchy. The system will be composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of primitive components is reached.

2. The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.

3. The linkages between components will change less than the components themselves.

4. Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements.

5. A complex system that works is invariably found to have evolved from a simple system that worked.

(Booch, 1991: 10-11)

2.5.1 Limitations of OOD While object-oriented design and programming have many strong points, some limitations still exist. The one limitation that dictated the use of a hybrid OOD was performance. The fact that objects make considerable use of smail methods (procedures) means there is a large overhead involved in function calls.(Booch, 1991: 216) Since speed was critical in the flight simulator design (as with any interactive graphic system), this limitation was something that required special attention.

Classic OOD dictates that all information passed between objects should be done using "get" and "set" methods. This requires a function call every time any object's attributes need to be used by another part of the program. To save the overhead of these calls, the needed attributes were made public and accessed directly by other objects using the C++ dot operator. This was especially important with objects like the State object, which contained many frequently accessed attributes describing the present state of the aircraft. Other small routines were defined "inline" to avoid using actual function calls.

2.6 Summary

As with almost all new systems, the *Virtual Cockpit* is a composite of many different disciplines. Flight simulation, software engineering, interactive computer graphics, and synthetic environment technologies all played a role in the development of this project.

III. System Design

3.1 Overview

3.1.1 Object-Oriented Design The entire project was designed in an object-oriented fashion. OOD models the physical world well, and that was used to our advantage. The three major components of the simulator became the highest level objects: the airplane, the virtual world, and the network. This thesis deals primarily with the airplane object. Capt McCarty designed and implemented the virtual world portion (McCarty, 1993), and Capt Sheasby implemented the distributed network aspects of the system (Sheasby, 1992).

OOD made the partitioning of the three major pieces very easy. Interfaces between the three objects were minimal which led to a very efficient working relationship among the team members as well as easily understood code. The simplicity of the design will also keep future maintenance problems to a minimum. The system was developed with yearly improvements and reconfigurations in mind. Interfaces were kept straightforward and made robust.

Figure 1 is a high-level representation of the simulator. All relationships between the objects are "is composed of" relationships. Communication between the objects is accomplished through the use of public attributes and, where communication is also across processor boundaries, "get" methods.

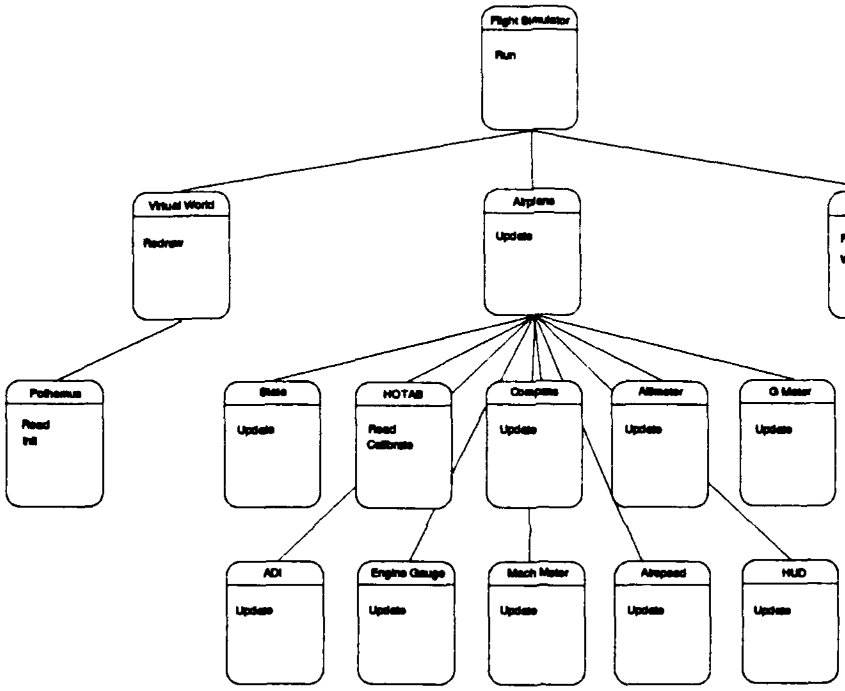


Figure 1. *Virtual Cockpit* Object Design

3.2 Frame Rate vs. Polygon Count

Realism requires both a high frame rate (15+ frames/sec), and a high polygon count to fully describe the objects in the scene. There is an inherent tradeoff between the two. The higher the polygon count, the more processor time required to render the polygons and the slower the frame rate. Since the *Virtual Cockpit* is a real-time system, the frame rate was deemed more important than the added realism gained from more polygons. A noticeable decline in the frame rate tends to make the simulated aircraft very difficult to handle, causing the operator to make constant overcorrections with the stick, and makes the entire simulation very unpleasant.

As a result, the airplane and cockpit descriptions were kept to a minimum number of polygons. The cockpit was not as detailed as it could have been, but more detail would have merely added to the "gee whiz" factor and not added any functionality to the simulation. Since the operator tends to look only at the cockpit instrument panel and out the canopy, attention was given to the gauges, HUD, and terrain features.

Texture-mapping added realism where it was possible to use it. In the cockpit, the faces of the instruments were texture-mapped to cut down on the polygon count. Since the instrument faces are static, relative to the cockpit, texture-mapping effectively eliminated 100 - 200 polygons. Dynamic parts of the instruments, like the needles, were modeled using very simple polygons.

3.3 Multiprocessor Utilization

In order to achieve a higher frame rate, the code was partitioned to run on three processors within the same machine. Since the Silicon Graphics Iris 440 VGXT is equipped with four processors, and the three objects were so clearly partitioned to start, it was a straightforward to move them to separate processors.

3.3.1 Interprocess Communication Paramount to a good multi-process design is eliminating as much of the communication between the processes as possible. If all communication can be eliminated, the processes have absolutely no dependence on each other and are truly autonomous. It will rarely happen in a program that two processes can be autonomous, but that was our goal in the design phase to lessen the communication between them. The three processes that make up the simulator are as autonomous as they could be made to be. Both the network and graphics rendering processes need only to take a "snapshot" of the aircraft state occasionally. That is the extent of the communication between them.

The sharing of data between the processes can be a serious problem without the use of critical section methods. The data may be in a non-determinate state at the time another process wishes to access it. To make sure that one, and only one, process is accessing the data at a time, locks or semaphores must be used as guards. Locks were chosen instead of semaphores for speed reasons. A semaphore will block when another process has control of the critical section. That means the process will be taken off the processor and not put back until the other process is clear of the critical section and the processor is free. If a critical section is guarded by a lock, the locked-out process will spin, keeping control of the processor, and continue checking the lock to see when the other process is through.

Figure 2 is an example of how two of the processes communicate by passing one variable between them. Only one of the processes may read or write var1 at a time.

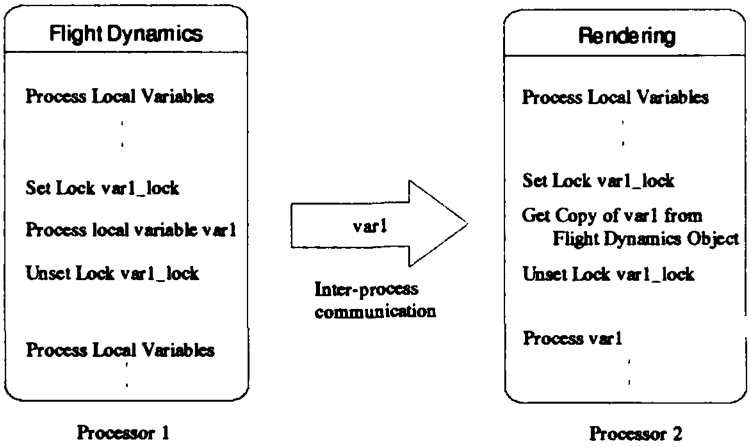


Figure 2. Interprocess communication

3.4 The Graphical Objects

In keeping with the two major components of the simulator, the graphical objects and the operational characteristics, the system was built using two different tools. The graphical objects, as well as the terrain and culture, were completed with a graphical modeling tool called Multigen and the operational characteristics were implemented with C++ code. Each piece of "virtual hardware" from the plane, down to the needle on the altimeter, is a separate Multigen file on disk.

Multigen uses a hierarchical database structure to model objects. By embedding external files in an object, a more complex object can be designed with little effort. The airplane is the highest level graphical object in the system. All the cockpit gauges are embedded in the airplane file. Likewise, the gauges are made up of differing amounts of other objects.

Typically, a gauge face is made up of a single square polygon with a texture map laid over it to give it details like numbers and hash marks. A needle may then be added by referencing an external needle file and positioning it on the face, anchored to the middle of the face. In our implementation, the external file is tagged as a dynamic object (it will move relative to the higher level object) by inserting a "1" in the first special attribute field of the higher level object database. The gauge is then externally referenced by the airplane object and visually placed on the airplane console.

The externally referenced files may be any name, but the order is important. A file named "objects.dyn," that is read at startup, contains the names of all the Multigen files (the names used within the simulator program, not the Multigen file name.) They must be in the same order they are referenced in the airplane file so the handles list used internally matches the order that the program reads the files in.

Each file tagged as dynamic must have a corresponding matrix associated with it so that the program can manipulate the object. The matrix contains the information about the six possible degrees of freedom any object may have; three degrees of orientation and 3-D position within the virtual world. The "objects.dyn" file establishes the link between the Multigen files and the matrices used to manipulate them. The link is the handle assigned to them with this file.

The matrices are manipulated in the Airplane object using the flight dynamics code for the airplane updates and the associated code for each instrument to manipulate the dynamic parts of the gauges. These matrices are passed across the processor boundary to the Virtual World object and used to render the Multigen files.

3.5 Flight Dynamics Model

While the flight dynamics model is arguably the most important part of the entire simulator, it is a problem that has been solved many times in the past. We concentrated almost entirely on the graphics solution to the problem and spent little time on the flight dynamics model. We used a simple off-the-shelf model based on only a small portion of the total operational envelope of the plane. Care was taken to provide the most intuitive interface to the model so as to facilitate its exchange with another more comprehensive model in the future.

The package that was decided on was from thesis work done at the Naval Post Graduate School by Capt Cooke (Cooke, 1991). His model was written in C, so it was relatively easy to integrate into C++ code.

I will highlight the most important aspects of the model and pay particular attention to those areas that are not straightforward. For a detailed discussion of the equations and techniques used, see Capt Cooke's thesis.

3.5.1 The State Object The State object defines everything about the airplane at given time. The math model is the "engine" of the object. It accepts inputs from both the simulated and real worlds and updates the state of the aircraft. The rest of the system relies on snapshots of the state variables for rendering and network communication.

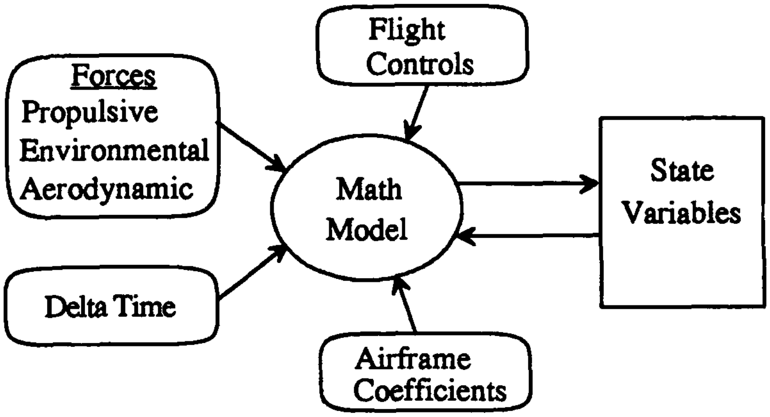


Figure 3. The Flight Dynamics Model

3.5.2 Interchangeable Aircraft One of the most compelling reasons to create a synthetic environment flight simulator is the ease with which attributes can be exchanged — both virtual hardware and operational characteristics. In order to take full advantage of this, the State object can be compiled with any distinct set of airframe coefficients. The coefficients are defined as constants and the definitions are kept in a separate file to be included in the State header file. To change the airframe coefficients, the needed definition file is included and the simulator is recompiled. This is exactly how other parts of the aircraft would be changed. If a new altimeter is needed, the new object (using the same interface) would be added to the Makefile and the system recompiled.

Two different coefficient files are included with the *Virtual Cockpit:* the A-4, and the Navion. More aircraft can be added to the collection.

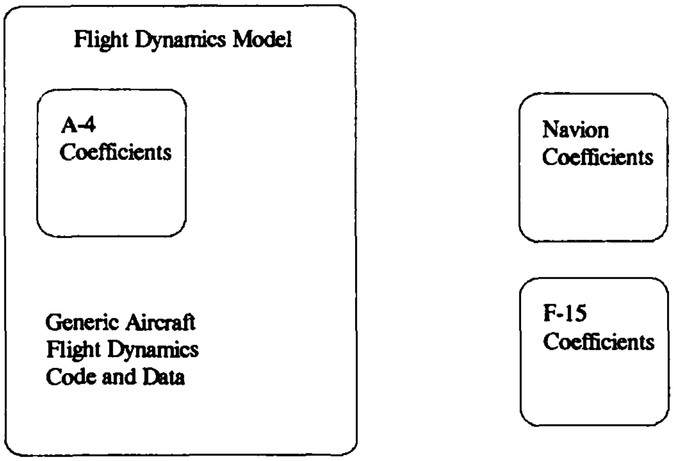


Figure 4. Flight Dynamics Model with Interchangeable Aircraft Coefficients

3.5.3 Coordinate Systems The flight dynamics model uses three different coordinate systems. A local, or aircraft, coordinate system, an aircraft world coordinate system, and finally, the *Virtual Cockpit* world coordinate system. The first two were part of the original model, and the last was a shell placed around the original model to abstract the complexities of the model away from the high level system. Instead of changing the inner workings of the model to accept the *Virtual Cockpit's* world coordinate system, it was decided to leave it as close as possible to its original structure to avoid having to modify the unfamiliar aerodynamic equations. The shell merely translates the final aircraft world coordinate system to the simulator's world coordinate system.

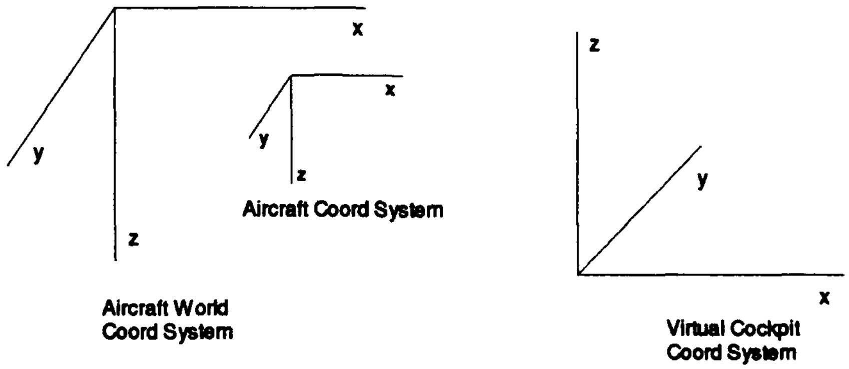


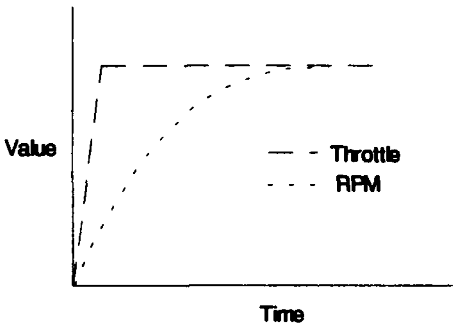
Figure 5. Coordinate Systems

As shown in Figure 5, the two original coordinate systems are identical in orientation. The local aircraft system's origin is defined to always be at the center of gravity of the airplane and gives the aircraft's position in terms of the change in orientation and position with respect to each axis and the last time step. With each time step, the incremental changes in orientation and position are added to the aircraft orientation and position in the world coordinate system. So while the nose of the aircraft remains oriented along the x-axis in the aircraft's local system, it changes with the orientation and position of the aircraft in the world system. The translation from the aircraft world system to the *Virtual Cockpit* system is a trivial rotation of 180° about the x-axis.

3.5.4 The Time Step Since the *Virtual Cockpit* does not run under a real-time operating system, adjustments were made to make the simulation act as much like a real-time system as possible. An observed problem with the original code was that it would "blow up" if the time step between calls to the flight dynamics model was too large. Several factors may cause the time step to vary radically, since Unix is doing many other tasks during the same time the *Virtual Cockpit* is running.

To make sure it is never too large, the delta time between State::Update() calls is divided into 20 smaller steps within the method. All 20 of the steps are looped through in one Update() call, covering the entire delta time. For example, if the time since the last Update() call is .01 seconds (delta time), the model will divide that into smaller .0005 second intervals and run through each loop of the math model with the smaller delta time.

3.5.5 Engine Spool-Up Time To better simulate a real jet engine, the engine RPM converges on the actual throttle position over time. This emulates the "spool-up" time for the turbine as shown in figure 6.



$$rpm = rpm + \frac{throttle - rpm}{2.0/deltatime}$$

Figure 6. RPM vs. Throttle Relationship

3.6 Hardware Interfaces

3.6.1 HOTAS The HOTAS (Hands-On Throttle and Stick) is a game-quality joystick and throttle that uses an RS232 serial interface. It was originally designed for use with a PC game card, but was modified by Thrust Master so the RS232 port of the Silicon Graphics workstation could be used. The speed was also increased from 4800 baud to 19,200 baud.

The HOTAS is a polled device. When a 0xFF (hexadecimal FF) is sent, it returns a string of 14 bytes representing the states of all switches and potentiometers at that time. See appendix B for details on the format of the data.

3.6.1.1 Throttles The throttles are dual, or split, throttles. Values for both the left and right throttles are available, so two distinct engines can be simulated if desired. The flight dynamics model used does not have the capability to model the aerodynamic peculiarities of two engines, so only one engine was modeled. The values for the left and right throttles are just added together and divided by two, resulting in the equivalent of one throttle.

3.6.1.2 Joystick The joystick in every airplane has a different feel to it. Some are very sensitive, while others are considered sloppy. In order to emulate as wide a variety as possible, I built a scaling factor into the joystick read routine. The stick returns two values: one for roll and one for pitch. The values received from the stick are between 0 and 255. This value is translated by the software interface to return from -1 to +1, with 0 at the middle of the deflection.

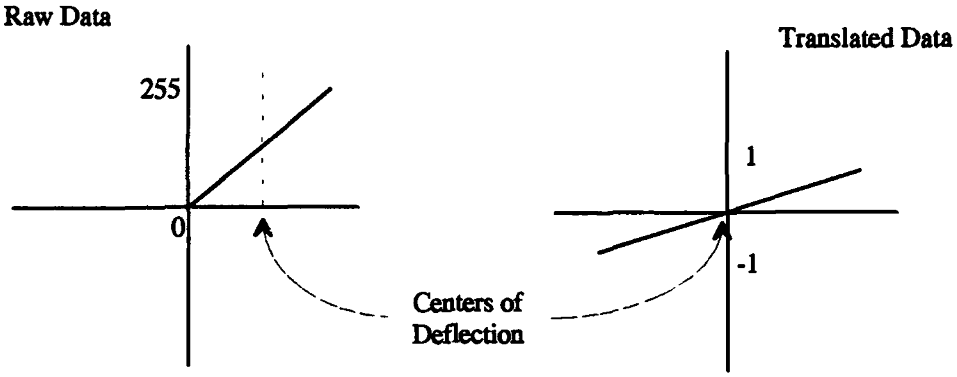


Figure 7. Raw and Translated Data from Joystick

This linear response produced a very sensitive stick. Only very slight movements of the stick were needed to produce a large deflection of the control surfaces. To minimize this sensitivity, and also to allow for the differences found in other aircraft, another parameter was added to the HOTAS class to define the sensitivity of the joystick. If this parameter is set to 1.0, the stick will exhibit the same linear characteristic it originally had. If the sensitivity is set to something greater than 1.0, it will become less sensitive. The option for a more sensitive stick is available by setting it to less than 1.0, but that is not recommended.

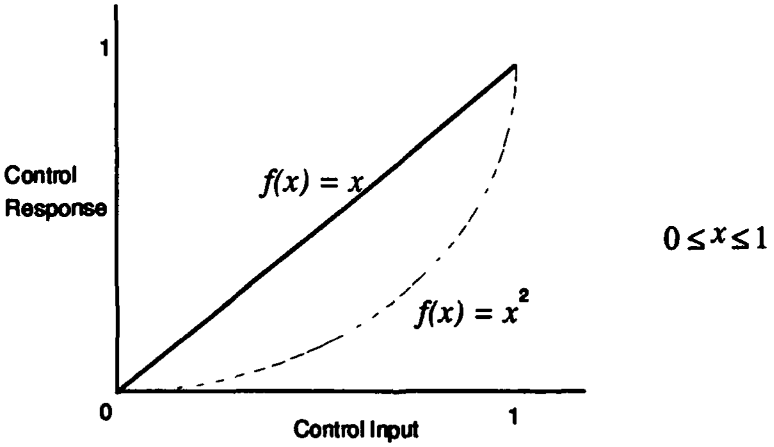


Figure 8. Joystick Sensitivity Scaling

Figure 8 shows how the scaling is accomplished. By raising the control input value to the power of SCALE, smaller stick deflections give a dampened response and the maximum deflections give an exaggerated response. Since most of the stick travel is slight in normal flying, this response is desired.

A "deadzone" is also incorporated into the stick interface. It allows the stick to travel within the limits of a square centered about the middle of the joystick and return a value of zero. If a SCALE of 2.0 or larger is used, the deadzone is not really needed because the dampened effect of the scaling effectively creates a deadzone about the center of deflection. Some amount of deadzone is necessary to avoid extraneous stick inputs while it is resting in the middle. A response is not wanted until the pilot actually moves the stick.

3.6.1.3 Rudder Pedals The HOTAS provides a value for a set of optional rudder pedals and the software interface reads and reports this value. The rudder pedals have not been received at the time of this writing. When they do arrive, it will take a minimum amount of effort to add them to the system.

3.5.1.4 Switches The throttles and stick have a number of push-button, toggle, and four-way switches on them. The software interface was designed to included the capability to use these switches, but time constraints allowed the implementation of only two of them.

The throttles do not have an afterburner (military power) detent, so I used a thumb switch on the right throttle to achieve the effect of an afterburner detent. When the throttles are greater than or equal to 98% of their maximum deflection, the thumb button will activate the afterburners. They will stay active until the throttles are moved below the 98% point.

The other switch used was the trim "hat" on top of the stick. Elevator trim can be adjusted by clicking the hat forward or backward. The trim adds an offset to the elevator value in the aerodynamics model. The hat supports aileron trim also, but aileron trim is rarely, if ever, used in aircraft, so it was not implemented.

3.6.2 Polhemus 3-Space Tracker In order to interface the HMD to the *Virtual Cockpit*, the Polhemus 3-Space Tracker was used to report the position and orientation of the user's head.

With the position and orientation of the head known, the scene can change to reflect the part of the virtual world toward which the user is looking.

This was a straightforward procedure, since the Polhemus interface had already been written and tested by previous students. It was left to simply read the values and adjust the position and orientation of the eyepoint and look direction accordingly.

To facilitate development without using the HMD, a class was created that uses the same interface the Polhemus uses but accepts input from the keyboard instead of the Polhemus. All six degrees of freedom can be input by using the arrow keys for orientation and some of the alphabetic keys for positioning. Please see the *Virtual Cockpit* manual in Appendix A for more information on the keys used.

3.7 Design Limitations

3.7.1 Hybrid OOD While a perfect object-oriented design was the goal, certain realities made this impossible. The initial design of the instrument objects were completely self-encapsulated and used inheritance, but neither concept survived to the final design.

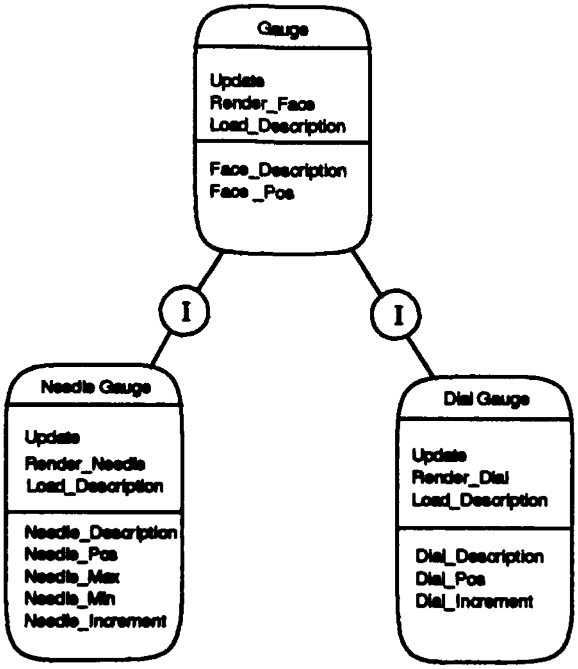


Figure 9. Original Gauge Design with Encapsulation and Inheritance

3.7.1.1 Object Encapsulation One would like an object to be completely self-sufficient. For example, an instrument should be able to update itself completely. It should possess a complete description of itself among its many attributes and should be able to render itself upon demand. If this were true, different instrument objects could be exchanged with a trivial amount of effort and nothing outside the object would need to know anything about how it worked.

True object-oriented programs are data driven. Since only one process on a Silicon Graphics workstation can utilize the graphics pipeline at a time, it was necessary to divide some parts of the program along lines of functionality, not data. With this limitation, the objects with graphical descriptions (like a gauge) could not be self-encapsulated, and were, in effect, split in half, with the graphics attributes and rendering functionality existing outside the object in a rendering object which was running on a different processor. The objects retained only the operational characteristics and the state of the image.

3.7.1.2 Inheritance Inheritance could be supported by creating simple base classes and adding only the complexity needed by each distinct type of instrument. The inheritance would have been very straightforward, as shown in figure 9. However, since the graphics file handle and rendering ended up being taken out of the gauge object, the only commonality left was the fact that they were gauges. What was left in the object, the state and distinct update methods, would not have made good use of inheritance. Since the goal was to develop a simple design, inheritance was not used just for the sake of using it.

3.7.2 Virtual World Technology Failing — The HUD All images within the program are polygon-based and stored in files. And while the graphics library bundled with the Silicon Graphics workstation provides a rich set of routines for drawing images dynamically, the simulator was designed (in keeping with virtual world design tenets) strictly for drawing already established files of polygons representing physical objects. This approach made it impossible to, say, draw a line in the environment that is constantly changing in length (changing orientation is no problem.) One is forced to anticipate all lengths the line will take on and keep separate files for each one of these lengths. The problem then is that of swapping the different lines in and out of the environment as the state of the image dictates. A significant change in the way graphical objects are defined and modeled, with tools like Multigen, will be needed to fully overcome this problem. Attributes like end-points will need to be dynamically alterable.

This design limitation was especially troublesome in designing the HUD. All other instruments had, at most, ten unique, external, dynamic graphics files embedded within them. Almost all gauges had just one external file (like a needle) that would simply change orientation when updated. The HUD, on the other hand, ended up with hundreds of dynamic components.

The HUD is fundamentally different from the gauges in that there are no "physical" moving parts. A real HUD's screen is made up completely of computer generated graphics. Ironically, the simulator design is much more adept at modeling physical hardware and has no inherent way of displaying simulated computer graphics. This was also the foremost problem with the design of the RADAR — a big enough problem to prohibit its implementation during this thesis cycle.

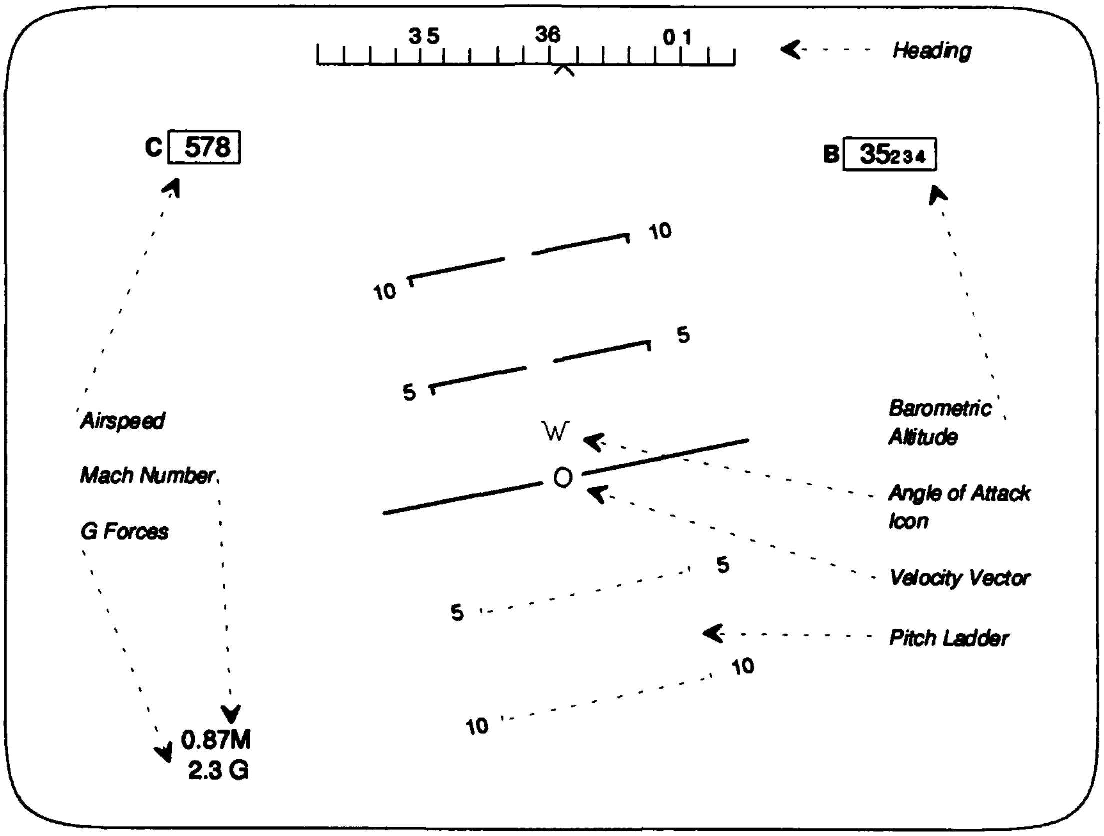


Figure 10. Head Up Display (HUD)

The HUD has several different fields of information displayed on it. Some information is presented with moving lines and icons, while the rest is displayed numerically. Icons, like the small wings that give angle of attack information, are as easy to manipulate as a needle on a standard gauge since they only change in position. The numeric fields were implemented by creating a separate file for each number. They all occupy the same position on the HUD when active, but only one number is active at a time. All others are translated off the HUD to a place where they will not be seen (like the center of the airplane.) While this approach works well, all the "numbers" still have to be manipulated with each pass through the HUD operational code. Every number is first moved off the HUD to clear it, then the one number that belongs there is translated back to the HUD.

Another technique is used for the pitch ladder. The ladder is broken into pieces representing all possible aircraft attitudes. Each separate ladder file moves through a range of 5° and then is swapped out for another file. This reduces the problem to that of moving a fixed object around on the HUD screen — an easy task. It requires 40 of these files, though, just to represent the pitch ladder.

The time required for the large of number of matrix manipulations required to do this turned out to be negligible, but the time required to handle them on the rendering side was noticeable. Each object still needed to be looked at, if only to decide that it was not in view and needed to be clipped away. And after all the clipping was done, there were still a considerable number of polygons on the HUD to be rendered.

3.8 Summary

The *Virtual Cockpit* was designed from the start in an object-oriented fashion. There were two reasons to use OOD:

1. OOD is an excellent technique to model a physical process or environment. It allows the attributes of the objects to drive the design and not the operational characteristics. The "is composed of" (ICO) relationship among objects worked very well in modeling an airplane where natural ICO relationships abound.

2. OOD makes the program much easier to understand and modify. Since this was primarily a research project and not a production version, we had to keep in mind that many more students will come after us to work on the code. If the original structure was intuitive, it would make their learning curve much smaller and foster better modifications.

The many semi-independent subsystems of the simulator and aircraft lent themselves nicely to decomposition and work division. Not discussed in this chapter were the contributions of Capt Sheasby and Capt McCarty who worked on the network connection and graphics rendering system, respectively.

IV. Results and Recommendations

4.1 Accomplishments

The goals of this thesis were achieved. The concept of a synthetic environment flight simulator was proved, and the system was designed and built in a fashion that makes modifications and additions easy.

The *Virtual Cockpit* was tested in a distributed environment with many other simulators during the months of October through December, 1992 (Zealous Pursuits 1-6) at the Institute for Defense Analysis in Washington, D.C. It was shown that a low-cost flight simulator can perform well in a distributed, multi-vehicle environment.

AFIT is now in a position to take this research much further. The OOD nature of the *Virtual Cockpit* ensures efficient expansions of the design and, at the same time, safe modifications to existing functionality.

4.2 Software Engineering Issues

This thesis was devoted primarily to expanding computer graphics research at AFIT, but we made use of many relatively new software engineering principles at the same time, so it is fitting to report on the results of that side of the project also. OOD, software reuse, and the Incremental-Build software process model were all used to reach our goal as productively as possible.

4.2.1 OOD Since OOD was such an integral part of the system design, it has already been mentioned several times. Suffice it to say, this relatively new program design technique was invaluable to the project and its contribution should continue to benefit those who come after us.

4.2.2 Software Reuse Software reuse was an important part of this project. Over the years, the AFIT Graphics Lab has developed many applications. A significant amount of software is now available for possible reuse. In particular, many low-level device driver interfaces have been developed as C++ classes.

For the most part, the reuse of these classes for the Polhemus, RS232 serial ports, and the SGI windowing system went smoothly. Hours of development time were saved by not having to "reinvent" these basic classes. The original authors generally took care to make the classes robust by adding methods to reconfigure the interface and/or operating mode of the device.

With only a week of work on the main driver and a simple flight dynamics model, the simulator was running. This was accomplished only because the window and joystick objects developed by previous students were reused. While not quite in its final form, the structure of the simulator did not change much between that initial version and the "final" version. More objects were added, some were exchanged for different ones, and some were made more complex, but interfaces did not change.

The one reuse failure encountered was that of a lack of thoroughness. A method was included to change an operating characteristic in one device class, yet when invoked, it did not change modes. After much time, it was discovered that the parameter was "hardwired" for one particular mode and there were no hooks back to the method designed to reconfigure it. Most likely a lack of time on the author's part contributed to the oversight. This points out the true enemy of reuse — the added overhead of designing and coding not just for the author's use, but also for future users.

4.2.3 Incremental-Build Since the *Virtual Cockpit* will most likely never be completed, an incremental-build development process seemed to be the wisest of the many software process models available to us. A primary objective of this process is to have *some* functionality as soon as possible.

OOD contributes greatly to the incremental-build process by allowing the developer to create object shells that have the full interface, but limited functionality. This enables the program to be built up very quickly. For example, after the driver, the next object developed was the flight dynamics (state) object. The interface was developed fully, but the math model was very simple — it only moved the plane in constant turns and incremental altitude changes. This allowed the other two team members to move quickly with the other parts of the program and not wait for the full math model to be integrated.

4.3 Multiprocessor Design

While the fact that we would eventually split the program into multiple processes was always kept in mind, the program was devel oped in a traditional sequential manner. When the time came to move segments of the program to different processors, parts of the system needed to be redesigned to accommodate this fundamentally different style of programming.

In retrospect, these are some questions that should have been answered before any code was written:

- How many distinct processes will there be?
- What, exactly, will the function of each process be?
- Will the processes communicate? If so, will it be by shared memory or message passing?
- How and where will the critical sections be implemented? (They need to be as small as possible.)
- Will the processes need to be synchronized, or can they run autonomously?

4.4 Recommendations for Future Improvements

4.4.1 Cockpit Layout In order to get the most realism for the frame rate, the entire inside of the cockpit should be texture-mapped. Texture-mapping the cockpit will require very accurate photos of the inside of a real cockpit. They should all be taken from the pilot's eye-point to give the correct perspective when applied to flat surfaces in the cockpit model and they will need to be taken in diffuse light (cloudy day) to avoid shadows.

4.4.2 Weapon Systems To be a complete military flight simulator, a weapon system of some sort is required. This component was not implemented during this thesis cycle due to a lack of time. All released weapons (missile or projectile) will assume their own identities within the distributed simulation environment. They may be completely autonomous objects, or be controlled from the aircraft that launched them. They will need to posses their own flight dynamics model and be able to read and write to the distributed network.

Below are some suggestions for beginning the design of a weapon system.

Weapon System	Process Control	Comments
Guns and Free-Falling Bombs	Autonomous	Both guns and free-falling bombs are ballistic projectiles, but with different initial velocity vectors. Each projectile could spawn an autonomous process which will remain active until it "strikes" something. In the case of gun projectiles, it may be wise to spawn only every tenth or so bullet to avoid overwhelming the network.
RADAR-Guided Missiles	Controlled	A RADAR-guided missile receives all its guidance from the launching aircraft. As long as the launching aircraft remains locked-on, and the target remains in the missile's G-limit envelope, the missile will close on the target. Messages will need to be passed to the autonomous missile process from the weapon system process. The missile can use the State object for its flight dynamics model with the appropriate airframe coefficients to emulate a missile's flight characteristics. Input from the launching aircraft can be input to the model via the stick interface.
Heat-Seeking Missiles	Autonomous	A heat-seeking missile will normally close on the greatest source of heat within its envelope of maneuver. Since the distributed network protocol does not include a field for amount of heat given off by an object, this will have to be approximated. The missile can be made to lock-on only to normally heat-radiating objects, but once the missile is launched, it will have to close on the same object, regardless of any change in heat within its envelope. This will preclude any use of heat-seeker avoidance tactics, like throttling down the engines or releasing flares.
Self-Guided RADAR Missiles	Autonomous	A self-guided RADAR missile locks-on to a target with its own built-in RADAR. These missiles will act exactly like the above heat-seeking missiles. The difference is that their characteristics are accurately modeled within the environment. Any avoidance tactic used to lose a self-guided RADAR missile will work.

Table 1. Weapon Systems Suggestions

4.4.3 RADAR There are two fundamentally different RADAR modes in today's multipurpose fighter-bombers: the standard and long-used air-to-air mode, and the newer, much more complex, air-to-ground mode.

The air-to-air mode would not be that difficult to implement. It would be similar to the HUD in that there would be many predefined objects available to display, but only a fraction of them would be on the screen at any one time. All possible aircraft RADAR profiles (which change with aircraft type, range, and cross-section) would need to be defined and available for use. With a cone of antenna illumination defining what the RADAR could detect, it would be a straightforward task to query the Network object for any needed information about aircraft within that cone. Since the Network object is running on a separate processor, this would require interprocess communication.

The air-to-ground mode would require a technique not yet used in the system. The air-to-ground RADAR gives a realistic two-dimensional display of the terrain within the cone of illumination. This requires the simulation of dynamic analog information using computer graphics on the RADAR screen — the same problem encountered with HUD, but much more difficult to solve. The HUD information could be very accurately emulated with digital representations in the form of incrementally changing predefined images, but it would be impossible to represent an infinite number of different terrain features with that approach.

Below are some possible ways to implement the RADAR. Whichever way it is done, the RADAR display generation should be an independent process. It will be a time-consuming function and may slow down the frame rate if it is done sequentially. Note that the *rendering* of the RADAR display and the actual *generation* of the display would not be in the same process as the design stands now. The implementation would be the same as that of the other aircraft systems which have separate image generation and rendering components.

Possibility	Implementation	Comments
Ray-tracing	A RADAR-like emulation can be achieved by using a common graphics solution — ray-tracing. Ray-tracing and RADAR both send a "ray" out from a point of interest to determine the distance to the closest physical point. Grayscale or color could be used to give distance/contour clues.	Unfortunately, full-blown ray-tracing is still too slow to do in real-time. If a minimal number of points were collected, though, this might be made to work fast enough for marginal terrain information. Vehicles could be placed on the terrain by using the fast air-to-air technique described above.
Texture-mapping	By using the same methods used in the generation of the pilot's view, a separate 2-D image of what the RADAR "sees" could be generated internally. The resulting 2-D image could be decomposed into a MIP map and displayed as a texture map on the RADAR display.	While texture maps can be created dynamically, it remains to be seen how quickly even the latest Silicon Graphics hardware can do it. This approach would probably be as slow as the ray-tracing method described above.
2-D window	The image could be created as above, in the texture-mapping method, but instead of creating a texture map, the resulting image would be displayed in a separate window placed within the existing pilot's view window. Since the window would be a 2-D entity, it would not exist within the virtual world. Instead, the window would have to repositioned with each frame to stay within the RADAR display screen defined in the 3-D cockpit.	Although it is not consistent with the spirit of synthetic environment technology, this approach should be fast enough to be useful. The pilot will surely notice the discontinuity in the cockpit environment when he moves his head more than a small amount. The window will not be resizable to give distance perspective and will keep the same shape regardless of the angle from which it is viewed.

Table 2. Possible RADAR Implementations

4.4.4 Data Glove With software already written by previous AFIT students, one could interface one or both of the pilot's hands into the virtual world. The VPL Data Glove has been used in several applications to give position, gesture, and motion information about the user's hand. As applied to the *Virtual Cockpit*, this would enable the pilot to see his hand(s) on the stick and throttle and allow him to use virtual switches in the cockpit. There are enough switches on the HOTAS to provide almost any input the pilot needs, but adding the ability to adjust a RADAR function or tune a radio with console buttons and switches would dramatically increase the realism.

4.4.5 Performance The *Virtual Cockpit* performed up to expectations in all but one area — the frame rate. The philosophy used during development was, "Make it work, *then* make it work faster." Unfortunately, there was time enough only to make it work. The frame rate is below 5 frames/sec, which is not fast enough to be practical.

From the testing that was done, it was determined that the system was graphics-bound. There are several possible ways to make the graphics pipeline more efficient, but that is beyond the scope of this thesis. See Capt McCarty's thesis for more details. (McCarty, 1993)

4.5 Conclusion

The advent of the synthetic environment simulator, has, in effect, given rise to a new genre of military exercises — those that require a large number of simulators to participate. The high costs of the high-end flight simulators prohibit the use of more than one or two of them in these types of exercises.

In the area of task training, the *Virtual Cockpit* has also been shown to be a practical alternative to flight simulators costing more than an order of magnitude more. While not realistic enough to teach a pilot to fly an airplane, a synthetic environment flight simulator is useful for training pilots in tasks such as searching for mobile ground targets, RADAR techniques, weapon systems training, multi-aircraft maneuvers, and many others.

Appendix A. Virtual Cockpit Manual

A.1 Configuration Files

- "calibrate.sim": This file will be generated if not found at start-up in the same directory as "sim". It contains the calibration information for the HOTAS and is read at start-up. It can be changed at start-up by anserwing "yes" to the calibration querry. Since each HOTAS returns different values for pitch, roll, and throttle posistion, this file maps the unique values onto a common range that the simulator will recognize. Each HOTAS should be recalibrated when first used in the system. The file should be used and not changed if the same HOTAS is being used. If the stick does not appear to be performing accurately (e.g. the plane is rolling when the stick is centered), restart the simulator and recalibrate — sometimes the HOTAS pots will drift.

- "init_pos.sim": This file will be generated if not found at start-up in the same directory as "sim". It will position the aircraft at any 3-D point in the synthetic environment. To begin at a different position, change the numbers in the file. The numbers represent, in order, x-position, y-position, z-position.

A.2 The HOTAS

All the the HOTAS connections must be made before power is applied. The stick must be connected to the throttles, and the throttles must be connected to the serial port on the computer. If in doubt, unplug the power and reconnect it after the other connections have been made. No harm will come to the system or HOTAS if any connections are made out of order — the HOTAS will just not work properly.

A.3 Afterburners

Since there is no afterburner detent on the throttle, a thumb switch is used to emulate it. To engage the afterburners, move both throttles to the maximum position, and press the lowest thumb switch on the right throttle. To disengage, move either throttle back from the maximum position. There will be a noticeable jump in power on the engine gauge when they are engaged or disengaged.

A.4 Elevator Trim

The elevator trim is adjusted with the "hat" switch on the very top of the joystick. To trim the nose down, click the "hat" forward. To trim the nose up, click the "hat" backward. Each click will trim the elevators by ±0.1°.

A.5 Quitting

To quit the program, type "q". Note: the mouse cursor must be in the *Virtual Cockpit* window for any keyboard inputs to be recognized.

Appendix B. HOTAS Technical Notes

The HOTAS is a Thrustmaster WCS Split Throttle design. It was originally designed for use with a PC game card but redesigned for us to use an RS232 serial interface. It is a polled device that runs at 19200 baud. There is very limited documentation available for the HOTAS. The designer at Thrustmaster is Frank Bouton and he can be reached at (503) 639-3200.

B.1 Data Format

Due to a bug in the hardware, the data format will come in two different formats — with no way of telling which format is in effect. There will always be 14 bytes returned after a 0xFF is sent to the port, but the format changes dynamically.

Format #1: FF, 12 bytes of data, FF.

Format #2: FF, FF, 12 bytes of data.

The algorithm used to read the packet overcomes the problem by checking the format on each read.

```
INDEX = 0
send 0xFF
wait for 0xFF
read 13 byte packet into buffer
if first byte in buffer == 0xFF  increment INDEX
xfer 12 bytes from buffer to switch variables using INDEX into buffer
```

The data in the buffer comes in two forms: byte or bit. The potentiometers (e.g.. roll, pitch) use a full byte, giving a value between 0 and 255, while the switches use only on bit of a byte giving only on/off information. See the HOTAS software documentation for details on the data fields.

It should also be noted that the switches are not consistent with respect to what represents pressed and not pressed. Some have 1 = pressed, while others have 0 = pressed. Again, see the software documentation for more details.

B.2 Calibration

The joysticks do not return uniform values for the limits of the potentiometers. Each joystick, therefor, needs to have its own mapping from min/max range to -1/1. This is done by recording the values at the extreme corners of travel and the center rest position. The following (Figure 11) is an example of a possible configuration.

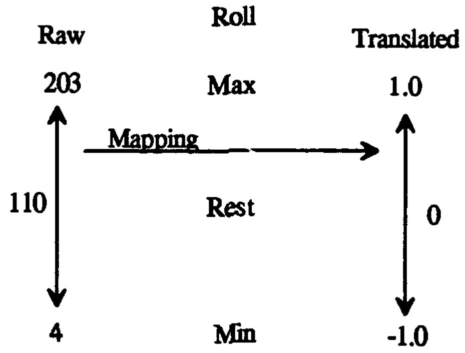


Figure 11. HOTAS Potentiometer Mapping

The min, max, and rest values are stored in a calibration file to be read at startup. Since each HOTAS is different, they should be calibrated when changed. Also, if the potentiometers drift and the joystick does not seem to be operating properly, a recalibration should be done by restarting the program and following the calibration instructions.

B.3 Device Connections

All the HOTAS connections must be made before power is applied. The stick must be connected to the throttles, and the throttles must be connected to the serial port on the computer. If in doubt, unplug the power and reconnect it after the other connections have been made. No harm will come to the system or HOTAS if any connections are made out of order — the HOTAS will just not work properly.

B.4 Special Cable

The RS232 connector on the HOTAS does not match the RS232 connector on the Silicon Graphics workstation. A special cable had to be made to interface the two. Only three wires are used.

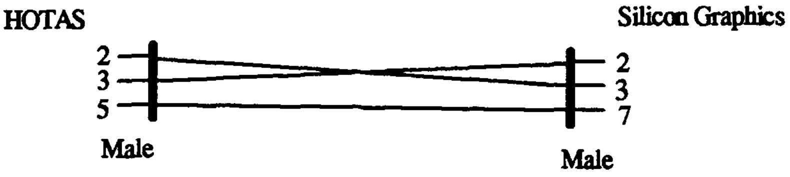


Figure 12. HOTAS to SG Cable Connections

Bibliography

Booch, Grady. *Object-Oriented Design With Applications.* Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.

----- and Michael Vilot, "The Design of the C++ Booch Components," *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA), 1990.* 1-11. New York:published in ACM SIGPLAN Notes vol 25, October 1990.

-----. "Object-Oriented Development," *IEEE Transactions on Software Engineering, 12:* 211-221 (February 1986).

-----. *Software Components With Ada.* Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.

Chung, J.C., et. al., "Exploring Virtual Worlds With Head-Mounted Displays," *Proceedings of the SPIE, 1083:* 42-52 (1989).

Clark, Jim, "Roots and Branches of 3-D," *Byte, 17:* 153-164 (May 1992).

Cooke, Capt Joseph ....need this info.

Dahn, Capt David A. *A Low Cost Part-Task Flight Training System: An Application of a Head Mounted Display.* MS thesis. AFIT/GCE/ENG/90D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

Filer, Capt Robert E.. *A 3-D Virtual Environment Display System.* MS thesis. AFIT/GCS/ENG/89D-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

Gerken, Capt Mark J.. *An Event Driven State Based Interface for Synthetic Environments.* MS thesis. AFIT/GCS/ENG/91D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

Lee, Kenneth J. and Michael S. Rissman. *An Object-Oriented Solution Example: A Flight Simulator Electrical System.* Prepared for the Electronic Systems Division, technical report number ESD-TR-89-5 by the Software Engineering Institute, February 1989. (AD-A219 190).

Lorimor, Capt Gary K.. Real-Time Display Of Time Dependent Data Using A Head-Mounted Display. MS thesis. AFIT/GE/ENG/89D-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

McCarty, Capt Dean M.I.. *Thesis Not Yet Titled.* MS thesis. AFIT/GCS/ENG/92D-XX. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, XXXXXX 1993.

Neyland, David. Notes from a meeting with the DARPA representative overseeing the development of the flight simulator held at the Air Force Institute of Technology (AU), Wright-Patterson AFB OH, April 12, 1992

Olson, Capt Robert A.. *Techniques to Enhance the Visual Realism of a Synthetic Environment Flight Simulator. MS thesis.* AFIT/GCS/ENG/91D-16. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

Platt, Capt Philip A. *Real-Time Flight Simulation and the Head-Mounted Display — An Inexpensive Approach to Military Pilot Training.* MS thesis. AFIT/GCS/ENG/90D-11 School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

Rebo, Capt Robert K.. *A Helmet-Mounted Virtual Environment Display System.* MS thesis. AFIT/GCS/ENG/88D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

Rolfe, J. M. *Flight Simulation.* New York: Cambridge University Press, 1986.

Sheasby, Capt Steven M.I.. *Thesis Not Yet Titled.* MS thesis. AFIT/GCS/ENG/92D-XX. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

Simpson, Capt Dennis Joseph. *An Application of the Object-Oriented Paradigm to a Flight Simulator.* MS thesis. AFIT/GCS/ENG/91D-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

Spicer, Capt Kelly L. *Mapping An Object-Oriented Requirements Analysis To A Design Archetecture That Supports Design And Component Reuse.* MS thesis. AFIT/GCS/ENG/90D-13. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

Sutherland, Ivan. "The Ultimate Display," *Proceedings of the IFIP Congress, 2:* 506-508 (1965).

Wang, Chu. "Designing for Interactive Performanc in a Virtual Laboratory," *Computer Graphics (ACM), no.2,* 39-40 (1990).

Wardin, Capt Charles L.. *Battle Management Visualization System.* MS theses. AFIT/GE/ENG/89D-56. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

Zyda, Michael. "Flight Simulators for Under \$100,000," *IEEE Computer Graphics and Applications, 8:* 19-27 (January 1988).

Vita

Captain John C. Switzer was born on 6 January 1962 in Seattle, Washington. He graduated from Inglemoor High School in Bothell, Washington in 1979. Following high school he enlisted in the Air Force and spent five years at Williams AFB, Arizona in the Weapons Control shop of the 425th Tactical Fighter Training Squadron. He was accepted into the Airman's Education and Commissioning Program in 1984 and went to the University of Washington. After his graduation with a Bachelors Degree in Electrical Engineering in 1987, he received his commission and was assigned to the 22nd Combat Communications Squadron at Patrick AFB, Florida where he headed the Engineering Division. After a tour of duty in the Persian Gulf during Desert Storm, he attended the Air Force Institute of Technology.

Permanent Address: 8304 NE 148th Pl, Bothell, WA 98011